Stupid Columnsort Tricks

Dartmouth College Department of Computer Science, Technical Report TR2003-444

Geeta Chaudhry
Thomas H. Cormen

Dartmouth College Department of Computer Science
{geetac, thc}@cs.dartmouth.edu

Contact author: Tom Cormen, 6211 Sudikoff Laboratory, Hanover, NH 03755.

Abstract

Leighton's columnsort algorithm sorts on an $r \times s$ mesh, subject to the restrictions that s is a divisor of r and that $r \geq 2s^2$ (so that the mesh is tall and thin). We show how to mitigate both of these restrictions. One result is that the requirement that s is a divisor of r is unnecessary; columnsort sorts correctly whether or not s divides r. We present two algorithms that, as long as s is a perfect square, relax the restriction that $r \geq 2s^2$; both reduce the exponent of s to $3/2$. One algorithm requires $r \geq 4s^{3/2}$ if s divides r and $r \geq 6s^{3/2}$ if s does not divide r. The other algorithm requires $r \geq 4^{3/2}$, and it requires s to be a divisor of r. Both algorithms have applications in increasing the maximum problem size in out-of-core sorting programs.

This research was supported in part by NSF Grant EIA-98-02068.

1 Introduction

The columnsort algorithm presented by Leighton in 1985 [Lei85] sorts N values on an $r \times s$ mesh, where $rs = N$, subject to three restrictions:

1. r must be even,
2. s must be a divisor of r (the *divisibility restriction*), and
3. $r \geq 2s^2$ (the *height restriction*).[1]

As long as these restrictions are obeyed, columnsort sorts the mesh into column-major order in eight steps. Steps 1, 3, 5, and 7 are all the same: sort each column. Each of steps 2, 4, 6, and 8 performs a fixed permutation on the mesh. That is, these permutations depend only on the values of r and s and not on the data being sorted. Therefore, columnsort is an *oblivious* sorting algorithm.

This paper presents the following results:

1. The divisibility restriction is unnecessary in the original columnsort algorithm.
2. By adding two more steps (one that is an oblivious permutation and one to sort each column), we can relax the height restriction by a factor of $\sqrt{s}/2$, to $r \geq 4s^{3/2}$, subject to the divisibility restriction, along with the new restriction that s is a perfect square (i.e., that $\sqrt{s}$ is an integer). We can also remove the divisibility restriction for this algorithm by tightening the height restriction to $r \geq 6s^{3/2}$.
3. By modifying columnsort to work with vertical slabs of $\sqrt{s}$ columns, we can create a different sorting algorithm with a height restriction of $r \geq 4s^{3/2}$, subject to the divisibility restriction and s being a perfect square.

These results were discovered during the course of the authors' work on using columnsort as the basis for out-of-core sorting [CCW01, CC02]. In various implementations, the column height is limited by the amount of memory per processor or the amount of memory throughout the entire parallel computer. In either case, the height restriction limits the problem size. Because problem sizes are large in out-of-core applications, a relaxation of the height restriction by a factor of $\Theta(\sqrt{s})$ can increase significantly the range of problem sizes that can be tackled. The authors have applied some of the ideas herein in current out-of-core sorting work [CHC03], and they plan to apply other ideas appearing in the present paper in future work.

The remainder of this paper is organized as follows. Section 2 reviews the original columnsort algorithm and gives a simple proof of its correctness. Section 3 shows that the divisibility restriction is unnecessary in the original algorithm. Section 4 presents the algorithm that adds the two steps (sorting and permuting) to relax the height restriction to $r \geq 4s^{3/2}$, and it shows that the divisibility restriction can be removed for this modified algorithm if $r \geq 6s^{3/2}$. Section 5 introduces the slab-based algorithm, proves its correctness, and shows that it has the height restriction $r \geq 4s^{3/2}$. Finally, Section 6 offers some concluding remarks.

2 Columnsort

This section describes Leighton's original columnsort algorithm. Using the 0-1 Principle, we also give a simple proof of its correctness.

[1]Leighton's original paper had a slightly more relaxed height restriction: $r \geq 2(s-1)^2$. The $r \geq 2s^2$ restriction matches better with proof methods in the present paper.

Given an $r \times s$ mesh, where r is even, s is a divisor of r, and $r \geq 2s^2$, columnsort operates in eight steps to sort the mesh all $N = rs$ values in column-major order. Each of steps 1, 3, 5, and 7 sorts each column. Each of steps 2, 4, 6, and 8 performs a specific oblivious permutation.

- As Figure 1 shows, treating the mesh as an $r \times s$ matrix, step 2 transposes the mesh and then reshapes the resulting $s \times r$ matrix back into an $r \times s$ mesh by taking each row of r entries and mapping it to a consecutive set of r/s rows.
- Step 4 performs the inverse of the permutation performed in step 2: treat each consecutive set of r/s rows as a single row in an $s \times r$ matrix, and transpose this matrix into the $r \times s$ mesh.
- Step 6 shifts each column down by $r/2$ positions. That is, the bottom half of each column moves to the top half of the next column to the right, and the top half of each column moves to the bottom half of the column. The evacuated top half of the leftmost column (column 0) is filled with the value $-\infty$. A new rightmost column (column s) is created, receiving the bottom half of column $s - 1$, and the bottom half of this new column is filled with the value ∞.
- Step 8 performs the inverse of the permutation performed in step 6: shift each column up by $r/2$ positions.

Proof of correctness

To show that columnsort really does sort, we first need to establish that it is oblivious. Clearly, the even-numbered steps perform oblivious permutations. The sorting method used in the odd-numbered steps might not be oblivious, however. As pointed out by Leighton [Lei92, p. 147], "No matter how the columns are sorted, the end result will look the same." Thus, we can imagine that an oblivious sorting method was used for the odd-numbered steps, knowing that we can substitute any sorting method of our choosing.

Because columnsort is oblivious, we can prove its correctness by means of the *0-1 Principle* [Lei92, pp. 141–142]:

> If an oblivious algorithm sorts all input sets consisting solely of 0s and 1s, then it sorts all input sets with arbitrary values.

When given a 0-1 input, we say that an area of the mesh is *clean* if it consists either of all 0s or all 1s. An area that might have both 0s and 1s is *dirty*. We shall show that steps 1–4 reduce the size of the dirty area to at most half a column and that steps 5–8 complete the sorting, assuming that the dirty area is at most half a column in size. As we read 0-1 values in a prescribed order within the mesh, a *$0 \rightarrow 1$ transition* occurs when a 0 is followed immediately by a 1; we define a *$1 \rightarrow 0$ transition* analogously.

Lemma 1 *Assuming a 0-1 input, after step 3, the mesh consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.*

Proof: As Figure 2(a) shows, after step 1, reading from top to bottom, each column consists of 0s followed by 1s. As we read a given column from top to bottom after step 1, there is at most one transition, and it is a $0 \rightarrow 1$ transition.

Step 2 turns each column into exactly r/s rows, shown in Figure 2(b). By the divisibility restriction, r/s is an integer. If we read the mesh in row-major order, any $1 \rightarrow 0$ transition occurs at the end of one row and the beginning of another. Therefore, read in row-major order, each set of consecutive r/s rows has at most the $0 \rightarrow 1$ transition from the corresponding column after step 1. Within each set of consecutive r/s

rows, the only row that may be dirty is the row containing the $0 \to 1$ transition. Since there are s such sets of rows, the mesh now has at most s dirty rows altogether.

Figure 2(c) shows the effect of step 3: moving the clean rows of 0s to the top rows of the mesh and the clean rows of 1s to the bottom rows of the mesh. The dirty rows, of which there are at most s, end up between the clean 0s and 1s. ∎

Lemma 2 *Assuming a 0-1 input, after step 4, the mesh consists of some clean columns of 0s on the left, some clean columns of 1s on the right, and a dirty area of size at most s^2 between them.*

Proof: By Lemma 1, the dirty area after step 3 is a set of consecutive rows numbering at most s. Since each row is s columns wide, the dirty area has size at most s^2. Step 4 permutes the clean rows of 0s at the top of the mesh to be clean columns of 0s on the left, the clean rows of 1s at the bottom of the mesh to be clean columns of 1s on the right, and the dirty area has at most s^2 entries between the clean areas. ∎

Corollary 3 *Assuming a 0-1 input, reading the mesh in column-major order after step 4, the dirty area is at most half a column in size.*

Proof: The height restriction $r \geq 2s^2$ is equivalent to $s^2 \leq r/2$. The dirty area has size at most s^2, and $r/2$ is the height of half a column. ∎

Lemma 4 *Assuming a 0-1 input, if the mesh, read in column-major after step 4, has a dirty area that is at most half a column in size, then steps 5–8 produce a fully sorted output.*

Proof: As Figure 3 shows, because the dirty area is at most half a column in size, it either fits in a single column or crosses from one column into the next. If the dirty area fits in a single column, then the sorting of step 5 cleans it, and steps 6–8 do not corrupt the sorted 0-1 output. If the dirty area is in two columns after step 4, then it is in the bottom half of one column and the top half of the next. Step 5 does not change this property, step 6 ensures that the dirty area resides in one column, step 7 cleans the dirty area, and step 8 moves all values back to where they belong. ∎

Theorem 5 *Columnsort sorts any input correctly.*

Proof: Immediate from columnsort being oblivious, the 0-1 Principle, Corollary 3, and Lemma 4. ∎

3 Removing the divisibility restriction

In this section, we show that columnsort, as described in Section 2, sorts correctly in the absence of the divisibility restriction. We continue to assume that r is even and at least $2s^2$.

As in Section 2, we rely on the 0-1 Principle. Because s might not be a divisor of r, however, we shall account for $1 \to 0$ transitions. The statement of the key lemma is similar to that of Lemma 1.

Lemma 6 *Assuming a 0-1 input but without the divisibility restriction, after step 3, the mesh consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.*

$$\boxed{\begin{array}{ccc} 0 & 9 & 18 \\ 1 & 10 & 19 \\ 2 & 11 & 20 \\ 3 & 12 & 21 \\ 4 & 13 & 22 \\ 5 & 14 & 23 \\ 6 & 15 & 24 \\ 7 & 16 & 25 \\ 8 & 17 & 26 \end{array}} \Rightarrow \boxed{\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \\ 12 & 13 & 14 \\ 15 & 16 & 17 \\ 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{array}}$$

Figure 1: The transpose-and-reshape operation of columnsort's step 2, shown for $r = 9$ and $s = 3$.

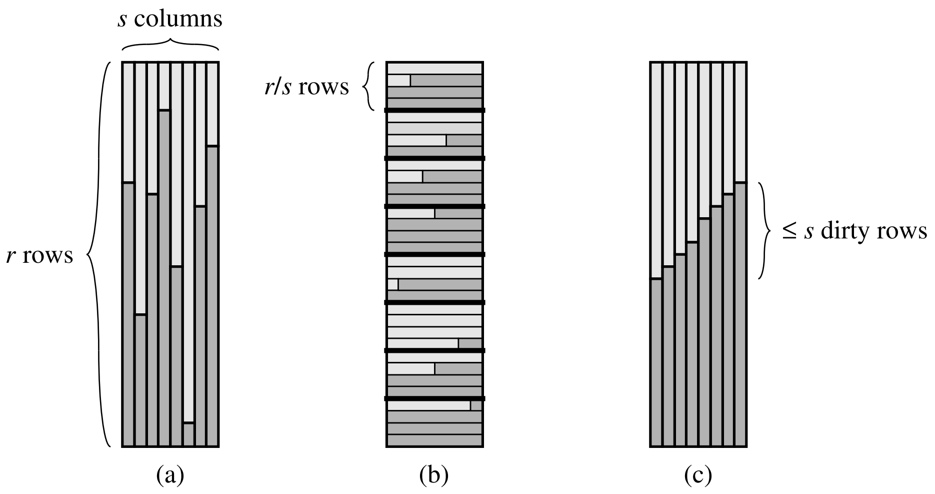


Figure 2: Steps 1–3 of columnsort, assuming a 0-1 input. 0s are lightly shaded and 1s are more darkly shaded. **(a)** After step 1, each column consists of 0s followed by 1s. **(b)** After step 2, each set of consecutive r/s rows has at most one $0 \to 1$ transition. There are at most s $0 \to 1$ transitions altogether. Heavy lines separate the rows formed from each column. **(c)** After step 3, the clean rows of 0s are at the top, the clean rows of 1s are at the bottom, and there are at most s dirty rows between them.

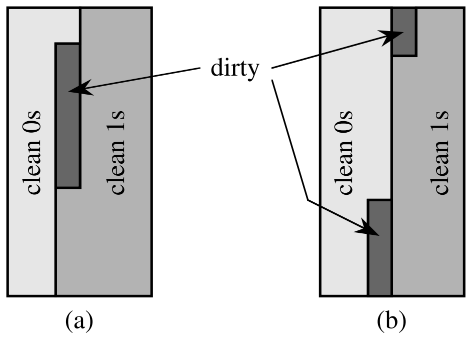


Figure 3: After step 4 the dirty area either **(a)** fits in a single column or **(b)** crosses from one column to the next.

Proof: As in the proof of Lemma 1, each column has at most one transition after step 1, and it is a $0 \to 1$ transition. Because s might not divide r, however, we can see $1 \to 0$ transitions within the rows after step 2. Figure 4 shows this possibility. There are still at most s $0 \to 1$ transitions, and because a $1 \to 0$ transition can occur only between pairs of consecutive columns after step 1, the number of $1 \to 0$ transitions is at most $s - 1$. Thus, as we read the mesh in row-major order after step 2, there are at most $2s - 1$ transitions of either type.

After step 2, let X be the set of dirty rows containing one $0 \to 1$ transition and no $1 \to 0$ transitions, Y be the set of dirty rows containing one $1 \to 0$ transition and no $0 \to 1$ transitions, and Z be the set of all other dirty rows (each of which must contain at least one $0 \to 1$ transition and at least one $1 \to 0$ transition).

We claim that $\max(|X|, |Y|) + |Z| \le s$. To prove this claim, note that every row of X and Z contains at least one $0 \to 1$ transition, and so $|X| + |Z| \le s$. Similarly, every row of Y and Z contains at least one $1 \to 0$ transition, and so $|Y| + |Z| \le s - 1$. If $\max(|X|, |Y|) = |X|$ then $\max(|X|, |Y|) + |Z| \le s$, and if $\max(|X|, |Y|) = |Y|$ then $\max(|X|, |Y|) + |Z| \le s - 1$. In either case, $\max(|X|, |Y|) + |Z| \le s$, thus proving the claim.

Now we examine the effect of step 3. As in the proof of Lemma 1, the clean rows of 0s move to the top and the clean rows of 1s move to the bottom. Let us consider pairs of rows in which one row of the pair is in X and the other row is in Y; there are exactly $\min(|X|, |Y|)$ such pairs of rows. When we pair them up and move rows of all 0s to the top and rows of all 1s to the bottom, we will have one of the following results:

$$\begin{array}{lccccc}
 & \text{more 0s than 1s} & & \text{more 1s than 0s} & & \text{equal 0s and 1s} \\
\text{from } X\text{:} & 00\cdots000011\cdots1 & & 00\cdots011111\cdots1 & & 00\cdots000111\cdots1 \\
\text{from } Y\text{:} & 11\cdots100000\cdots0 & & 11\cdots111100\cdots0 & & 11\cdots111000\cdots0 \\
 & \Downarrow & \text{or} & \Downarrow & \text{or} & \Downarrow \\
 & 00\cdots000000\cdots0 & & 00\cdots011100\cdots0 & & 00\cdots000000\cdots0 \\
 & 11\cdots100011\cdots1 & & 11\cdots111111\cdots1 & & 11\cdots111111\cdots1 \\
 & \text{clean row of 0s} & & \text{clean row of 1s} & & \text{clean rows of 0s and 1s}
\end{array}$$

In any case, at least one clean row is formed, and so at least $\min(|X|, |Y|)$ new clean rows are created. These clean rows go to the top (if 0s) and bottom (if 1s) of the mesh.

The number of dirty rows remaining is at most $|X| + |Y| - \min(|X|, |Y|) + |Z|$. Observing that $|X| + |Y| - \min(|X|, |Y|) = \max(|X|, |Y|)$, we see that the number of dirty rows remaining is at most $\max(|X|, |Y|) + |Z|$, which, by our earlier claim, is at most s. ■

From here, we can use the same technique as in Section 2 to prove the following theorem:

Theorem 7 *Even without the divisibility restriction, columnsort sorts any input correctly.* ■

4 Relaxing the height restriction by subblock distribution

Here we show how to relax the height restriction from $r \ge 2s^2$ to $r \ge 4s^{3/2}$, so that the mesh need not be quite so tall and thin in order for a columnsort-like algorithm to work. Our technique, inspired by the Revsort algorithm of Schnorr and Shamir for sorting on a square mesh [SS86], will be to add two more steps after step 3. After we describe the modified columnsort algorithm and prove that it works, we will show that we can additionally eliminate the divisibility restriction, but at the cost of a slightly tighter height restriction: $r \ge 6s^{3/2}$.

We will be working with $\sqrt{s} \times \sqrt{s}$ subblocks of the mesh, where each subblock is a contiguous set of $\sqrt{s}$ rows and $\sqrt{s}$ columns. Subblocks are aligned to the mesh, meaning that the indices of the top row and leftmost column of each subblock must be multiples of $\sqrt{s}$. We need $\sqrt{s}$ to be an integer, and so we require s to be a perfect square. For the moment, we also require the divisibility restriction and the height restriction that $r \geq 4s^{3/2}$.

We add the following steps between steps 3 and 4 of columnsort:

- Step 3.1 performs any permutation that moves all the values in each $\sqrt{s} \times \sqrt{s}$ subblock into all s columns and that does not depend on the data being sorted.
- Step 3.2 sorts each column.

We refer to the resulting algorithm, which is oblivious, as *subblock columnsort*.

There are several permutations that accomplish the goal of step 3.1. One option is to adapt the permutation proposed by Schnorr and Shamir [SS86], so that for $i = 0, 1, \ldots, r-1$, the ith row is cyclically rotated by $\text{rev}(i)$ bits to the right, where $\text{rev}(i)$ is the bit reversal of the $\lg s$ least significant bits of i.

An alternative permutation that accomplishes the goal of step 3.1 is the following. It requires both r and s to be powers of 2. (In fact, since s must be a perfect square, it should be a power of 4.) Each entry of the mesh has a row number, expressed in $\lg r$ bits, and a column number, expressed in $\lg s$ bits. To determine where the value in a given mesh entry goes, exchange the $\lg \sqrt{s}$ least significant bits of the row number with the $\lg \sqrt{s}$ most significant bits of the column number. To see that this permutation distributes all values in each $\sqrt{s} \times \sqrt{s}$ subblock into all s columns, note that within any subblock, no two row numbers have the same least significant $\lg \sqrt{s}$ bits, and the same holds for column numbers. Thus, for any two entries within a given subblock, the bits that form the column number they map to must differ in at least one place. (Incidentally, this permutation has the desired distribution property even if the $\sqrt{s} \times \sqrt{s}$ subblock does not begin at a row or column index that is a multiple of $\sqrt{s}$.)

Correctness of subblock columnsort

We shall show that as long as $r \geq 4s^{3/2}$, the dirty area after step 4 is at most half a column in size. As before, the 0-1 Principle and Lemma 4 will complete the proof of correctness. We start with the following lemma, which states another property that even the original columnsort algorithm has.

Lemma 8 *Assuming a 0-1 input and assuming that the divisibility restriction holds, after step 3, the line dividing the 0s and 1s is monotonic in the sense that, as shown in Figure 5, it goes from left to right and bottom to top, never turning back to the left and never turning back toward the bottom.*

Proof: Because step 3 sorts the columns, we cannot see any 0 below a 1 within a given column. Thus, the dividing line cannot turn back to the left.

To see that the dividing line cannot turn back toward the bottom, it suffices to show that after step 2, each column has at least as many 0s as the column immediately to its right. In order for a column to have fewer 0s than the column to its right, there would have to be some row in which these two columns exhibited a $1 \rightarrow 0$ transition. But due to the divisibility restriction, there are no $1 \rightarrow 0$ transitions within rows after step 2. ■

The following lemma uses an argument similar to one that was first made by Schnorr and Shamir.

Lemma 9 *Assuming a 0-1 input, after step 3.1 of subblock columnsort, the number of 0s in any two columns differs by at most $2\sqrt{s}$.*

Proof: As Lemma 1 shows, after step 3 (and before step 3.1), the dirty area is confined to an area that is $s \times s$. Referring to Figure 5, and noting that subblocks are $\sqrt{s} \times \sqrt{s}$ in size, the dirty area is confined to an array of subblocks that is at most $\sqrt{s} + 1$ high and at most $\sqrt{s}$ wide. Because the dividing line between 0s and 1s is monotonic, it passes through at most $2\sqrt{s}$ subblocks. In other words, all but $2\sqrt{s}$ subblocks are clean.

Step 3.1 distributes each subblock to all s columns. The clean subblocks distribute their 0s or 1s uniformly to each column, and thus the difference between the number of 0s in any two columns is at most the number of dirty subblocks: $2\sqrt{s}$. ■

Lemma 10 *Assuming a 0-1 input, after step 4 of subblock columnsort, the mesh consists of some clean columns of 0s on the left, some clean columns of 1s on the right, and a dirty area of size at most* $2s^{3/2}$ *between them.*

Proof: By Lemma 9, after sorting each column in step 3.2, the dirty area is confined to an area that is at most $2\sqrt{s}$ rows high and s columns wide. Thus, after step 3.2, the dirty area has size at most $2s^{3/2}$. As in the proof of Lemma 2, after step 4, there are clean columns of 0s on the left, clean columns of 1s on the right, and the dirty area is between them, but now of size at most $2s^{3/2}$. ■

Theorem 11 *Assuming that the divisibility restriction holds, s is a perfect square, and $r \geq 4s^{3/2}$, subblock columnsort sorts any input correctly.*

Proof: By Lemma 10, after step 4, the dirty area has size at most $2s^{3/2}$. Since $r \geq 4s^{3/2}$, we have that $2s^{3/2} \leq r/2$, and so the dirty area is at most half a column in size. We complete the proof by applying Lemma 4 and noting that because subblock columnsort is oblivious, the 0-1 Principle applies. ■

Removing the divisibility restriction from subblock columnsort

We next show that if we tighten the height restriction for subblock columnsort by 50%, so that $r \geq 6s^{3/2}$, then the divisibility restriction is unnecessary.

When we remove the divisibility restriction, there may be $1 \rightarrow 0$ transitions within rows after step 2. After step 3, therefore, there may be $1 \rightarrow 0$ transitions within rows. Figure 6 shows the dividing line between 0s and 1s after step 3: it is no longer monotonic like in Figure 5. The dividing line still cannot turn back to the left (since step 3 sorts each column), but it may turn back toward the bottom. The result is the "skyline" appearance shown in Figure 6.

The key to relaxing the height restriction in subblock columnsort was showing that the boundary between 0s and 1s passes through a limited number of subblocks. When the boundary can both rise and fall, as in the skyline pattern of Figure 6, one might suspect that each peak and each valley could be only one column wide and there might be very tall peaks and very deep valleys; in this case, the boundary could pass through all $s + \sqrt{s}$ subblocks in a region $\sqrt{s} + 1$ subblocks high and $\sqrt{s}$ subblocks wide. The following lemma shows that this scenario is overly pessimistic: the total length of the rising and falling edges is limited by $2s - 1$.

Lemma 12 *Assuming a 0-1 input but without the divisibility restriction, after step 3 of subblock columnsort, the line dividing the 0s and 1s has a total length of rising and falling edges that is at most* $2s - 1$.

Proof: A rising edge occurs whenever there is a $0 \rightarrow 1$ transition within a row, and a falling edge occurs whenever there is a $1 \rightarrow 0$ transition within a row. For $j = 0, 1, \ldots, s - 2$, let x_j denote the number of

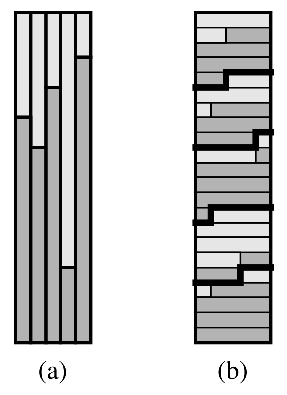


Figure 4: Steps 1 and 2 when s does not divide r. **(a)** After step 1. **(b)** Within the rows after step 2, there is at most one $0 \to 1$ transition for each column after step 1, and there is at most one $1 \to 0$ transition for each pair of consecutive columns. Heavy lines separate the values originally from each column.

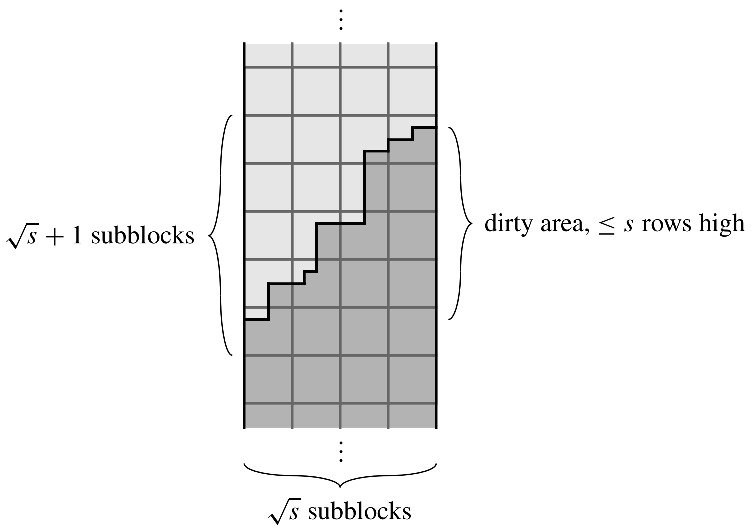


Figure 5: The monotonicity of the dividing line between the 0s and 1s after step 3. The dividing line passes through at most $2\sqrt{s}$ subblocks.

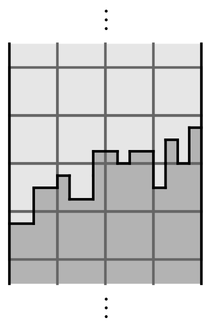


Figure 6: The "skyline" shape of the dividing line between the 0s and 1s after step 3 when the divisibility restriction is removed. The total length of the rising and falling edges is bounded by $2s - 1$.

$0 \to 1$ transitions between columns j and $j+1$ after step 2, and x'_j denote the number of $0 \to 1$ transitions between columns j and $j+1$ after the columns are sorted in step 3. Define y_j and y'_j similarly for $1 \to 0$ transitions. Letting $x' = \sum_{j=0}^{s-2} x'_j$ and $y' = \sum_{j=0}^{s-2} y'_j$, it suffices to show that $x' + y' \le 2s - 1$.

We claim that $y'_j - x'_j = y_j - x_j$ for $j = 0, 1, \ldots, s-2$. To see why, let z_j equal the number of 0s in column j, for $j = 0, 1, \ldots, s-1$; note that z_j does not change due to the sorting in step 3. Because each $1 \to 0$ transition causes column $j+1$ to have one more 0 than column j and each $0 \to 1$ transition causes column $j+1$ to have one fewer 0 than column j, we have that $z_{j+1} = z_j + y_j - x_j$ and also $z_{j+1} = z_j + y'_j - x'_j$ for $j = 0, 1, \ldots, s-2$. Subtracting z_j from each formula gives $y'_j - x'_j = z_{j+1} - z_j = y_j - x_j$, which proves the claim.

Next we claim that after step 3, for $j = 0, 1, \ldots, s-2$, at least one of x'_j and y'_j must be 0. We prove this claim by contradiction: suppose that both x'_j and y'_j are positive. Then in columns j and $j+1$, there is a row with a $0 \to 1$ transition and a row with a $1 \to 0$ transition. If the row with the $0 \to 1$ transition is the higher of the two, then column $j+1$ has a 1 above a 0, which cannot occur since it has been sorted in step 3. If instead the row with the $1 \to 0$ transition is the higher of the two rows, then column j has a 1 above a 0, which again cannot occur. Since we cannot have both a $0 \to 1$ transition and a $1 \to 0$ transition within the same row, at least one of x'_j and y'_j must be 0, thus proving the claim.

Noting that all x_j and y_j are nonnegative, $x'_j = 0$ implies $y'_j = y_j - x_j \le y_j$, and $y'_j = 0$ implies $x'_j = x_j - y_j \le x_j$. In either case, we see that $x'_j \le x_j$ and $y'_j \le y_j$ for all $j = 0, 1, \ldots, s-2$. Thus, we have

$$\begin{aligned} x' + y' &= \sum_{j=0}^{s-2} x'_j + \sum_{j=0}^{s-2} y'_j \\ &\le \sum_{j=0}^{s-2} x_j + \sum_{j=0}^{s-2} y_j \,. \end{aligned}$$

As we saw in the proof of Lemma 6, $\sum_{j=0}^{s-2} x_j \le s$ and $\sum_{j=0}^{s-2} y_j \le s-1$. Therefore, $x' + y' \le s + (s-1) = 2s - 1$. ■

Corollary 13 *Assuming a 0-1 input but without the divisibility restriction, after step 3.1 of subblock columnsort, the number of 0s in any two columns differs by at most* $3\sqrt{s}$.

Proof: The proof is the same as that of Lemma 9, but with the change that because the total length of the rising and falling edges of the dividing line between 0s and 1s is at most $2s - 1$, it passes through at most $3\sqrt{s}$ subblocks. ■

Corollary 14 *Assuming a 0-1 input but without the divisibility restriction, after step 4 of subblock columnsort, the mesh consists of some clean columns of 0s on the left, some clean columns of 1s on the right, and a dirty area of size at most* $6s^{3/2}$ *between them.*

Proof: Analogous to the proof of Lemma 10. ■

Theorem 15 *Assuming that s is a perfect square and that* $r \ge 6s^{3/2}$, *but without the divisibility restriction, subblock columnsort sorts any input correctly.*

Proof: Analogous to the proof of Theorem 11. ■

5 Relaxing the height restriction by division into vertical slabs

In Section 4, we relaxed the height restriction from $r \geq 2s^2$ to $r \geq 4s^{3/2}$. In this section, we present another columnsort-based algorithm that also has a height restriction of $r \geq 4s^{3/2}$ and obeys the divisibility restriction. Although this algorithm will have 11 steps, two of these steps are fixed permutations that can be composed into one permutation.

The algorithm is based on partitioning the mesh into several vertical "slabs," where we define a *k-slab* as a set of k consecutive columns, with the leftmost column in the slab at an index that is a multiple of k. (The inspiration for using slabs comes from the work of Marberg and Gafni [MG88] for sorting on a square mesh.) When forming slabs, we assume that k is a divisor of s (and hence, by the divisibility restriction, a divisor of r). For a given value of k, there are s/k k-slabs; since k is a divisor of s, so is s/k. The optimal value of k will turn out to be $k = \sqrt{s}$—so that s will need to be a perfect square—but for now we shall think in terms of general values of k.

This algorithm requires two new operations, both of which are independent of the data being sorted:

- A *k-slabpose*, shown in Figure 7(a), is a transpose and reshape operation, but within each k-slab. Just as step 2 of the original columnsort algorithm transposes the mesh and reshapes it back into an $r \times s$ arrangement, a k-slabpose transposes within each k-slab and reshapes it back into an $r \times k$ configuration.
- A *k-shuffle*, shown in Figure 7(b), is a permutation of the s columns of the mesh in which we first take in order all columns whose indices are congruent to 0 modulo k, then take in order all columns whose indices are congruent to 1 modulo k, then 2 modulo k, and so on. More precisely, to determine which index column j maps to, let $j = lk + m$, where $0 \leq l < s/k$ and $0 \leq m < k$; then column j maps to index $l + ms/k$.

With these two new operations, we present the algorithm, which we call *slabpose columnsort*. Start by choosing any value of $k \leq \sqrt{s}$. Then perform the following 11 steps:

- Step 1 sorts each column.
- Step 2 performs a k-slabpose.
- Step 3 sorts each column.
- Step 4 is a k-shuffle.
- Step 5 performs an (s/k)-slabpose.
- Steps 6–11 are the same as steps 3–8 of the original columnsort algorithm.

This algorithm is oblivious. Note that because steps 4 and 5 are consecutive and both perform fixed permutations, they can be replaced by a single step that performs the composition of the k-shuffle and (s/k)-slabpose permutations. The resulting algorithm would have 10 steps rather than 11. To ease understanding, however, we shall focus on the 11-step formulation in the remainder of this section.

Correctness of slabpose columnsort

In our now-familiar method of proving correctness, we shall assume a 0-1 input and show that after step 7 of slabpose columnsort (which corresponds to step 4 of original columnsort), the dirty area is at most half a column in size.

Lemma 16 *Assuming a 0-1 input, after step 3 of slabpose columnsort, each k-slab consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most k dirty rows between them.*

Proof: Steps 1–3 of slabpose columnsort are like steps 1–3 of original columnsort, but run on each k-slab independently. The proof then follows from Lemma 1. ■

Figure 8(a) shows the mesh after step 3. The dirty rows in one k-slab bear no relation to the dirty rows in any other k-slab, and so overall it is possible that up to s rows in the mesh are still dirty. As the following lemma shows, steps 4 and 5 reduce the number of dirty rows in the entire mesh.

Lemma 17 *Assuming a 0-1 input, after step 5 of slabpose columnsort, at most $(s/k)(\lceil k^2/s \rceil + 1)$ rows of the mesh are dirty.*

Proof: As Figure 8(b) shows, the k-shuffle of step 4 has the effect of creating k (s/k)-slabs. For $j = 0, 1, \ldots, k-1$, the dirty part of the jth column is confined to the same set of k rows in each of the (s/k)-slabs. Since s/k is a divisor of r, the (s/k)-slabpose operation of step 5 permutes each column within an (s/k)-slab into a set of $r/(s/k) = rk/s$ consecutive rows within the same (s/k)-slab. Figure 8(c) illustrates the result. Because the jth column of each (s/k)-slab forms the jth set of rk/s consecutive rows, and the dirty part of the jth column is confined to the same set of rows in each of the (s/k)-slabs, we see that within each set of rk/s consecutive rows, the dirty rows are confined to the same set of rows.

Next we determine how many rows within each set of rk/s consecutive rows are dirty. Because the dirty rows align among (s/k)-slabs, we need examine just a single (s/k)-slab. Prior to step 5, any given column of an (s/k)-slab has a dirty area at most k rows high. When these k values are moved into rows of width s/k, they fall into at most $\lceil k/(s/k) \rceil + 1 = \lceil k^2/s \rceil + 1$ rows.

Since there are s/k sets of rk/s consecutive rows, the total number of dirty rows after step 5 is at most $(s/k)(\lceil k^2/s \rceil + 1)$. ■

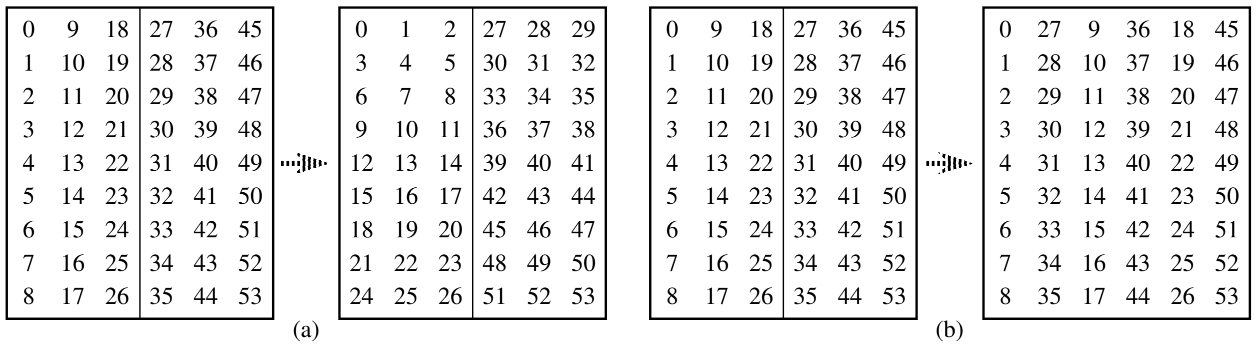
Theorem 18 *As long as k is chosen as a divisor of s, s is a divisor of r, and $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$, slabpose columnsort sorts correctly.*

Proof: If we assume a 0-1 input, then by Lemma 17, there are at most $(s/k)(\lceil k^2/s \rceil + 1)$ dirty rows after step 5. After step 6 (which sorts each column), there are clean rows of 0s at the top of the mesh, clean rows of 1s at the bottom, and at most $(s/k)(\lceil k^2/s \rceil + 1)$ dirty rows between them. Since the dirty area is at most s columns wide, it is confined to an area of the mesh of size $(s^2/k)(\lceil k^2/s \rceil + 1)$. After step 7, which is a full inverse transpose, when we read the mesh in column-major order, the dirty area is confined to at most $(s^2/k)(\lceil k^2/s \rceil + 1)$ consecutive entries. By Lemma 4, as long as this dirty area is at most half a column in size, the final four steps produce a sorted 0-1 output. This bound on the dirty area's size—$(s^2/k)(\lceil k^2/s \rceil + 1) \leq r/2$—is equivalent to the condition $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$ in the theorem statement. Noting that slabpose columnsort is oblivious and applying the 0-1 Principle completes the proof. ■

Corollary 19 *There is an implementation of slabpose columnsort that has the height restriction $r \geq 4s^{3/2}$ and the restriction that s is a perfect square.*

Proof: If we choose $k = \sqrt{s}$, then k is a divisor of s, and the height restriction $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$ becomes $r \geq 4s^{3/2}$. We need s to be a perfect square so that $\sqrt{s}$ is an integer. ■

Indeed, the height restriction of $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$ is minimized when we choose $k = \sqrt{s}$. To see why, let us consider separately candidate values of k that are at most $\sqrt{s}$ and greater than $\sqrt{s}$. If $k \leq \sqrt{s}$,



0	9	18	27	36	45
1	10	19	28	37	46
2	11	20	29	38	47
3	12	21	30	39	48
4	13	22	31	40	49
5	14	23	32	41	50
6	15	24	33	42	51
7	16	25	34	43	52
8	17	26	35	44	53

0	1	2	27	28	29
3	4	5	30	31	32
6	7	8	33	34	35
9	10	11	36	37	38
12	13	14	39	40	41
15	16	17	42	43	44
18	19	20	45	46	47
21	22	23	48	49	50
24	25	26	51	52	53

(a)

0	9	18	27	36	45
1	10	19	28	37	46
2	11	20	29	38	47
3	12	21	30	39	48
4	13	22	31	40	49
5	14	23	32	41	50
6	15	24	33	42	51
7	16	25	34	43	52
8	17	26	35	44	53

0	27	9	36	18	45
1	28	10	37	19	46
2	29	11	38	20	47
3	30	12	39	21	48
4	31	13	40	22	49
5	32	14	41	23	50
6	33	15	42	24	51
7	34	16	43	25	52
8	35	17	44	26	53

(b)

Figure 7: k-slabpose and k-shuffle operations, shown for $k = 3$ on a 9×6 mesh. **(a)** A k-slabpose operation. **(b)** A k-shuffle operation.

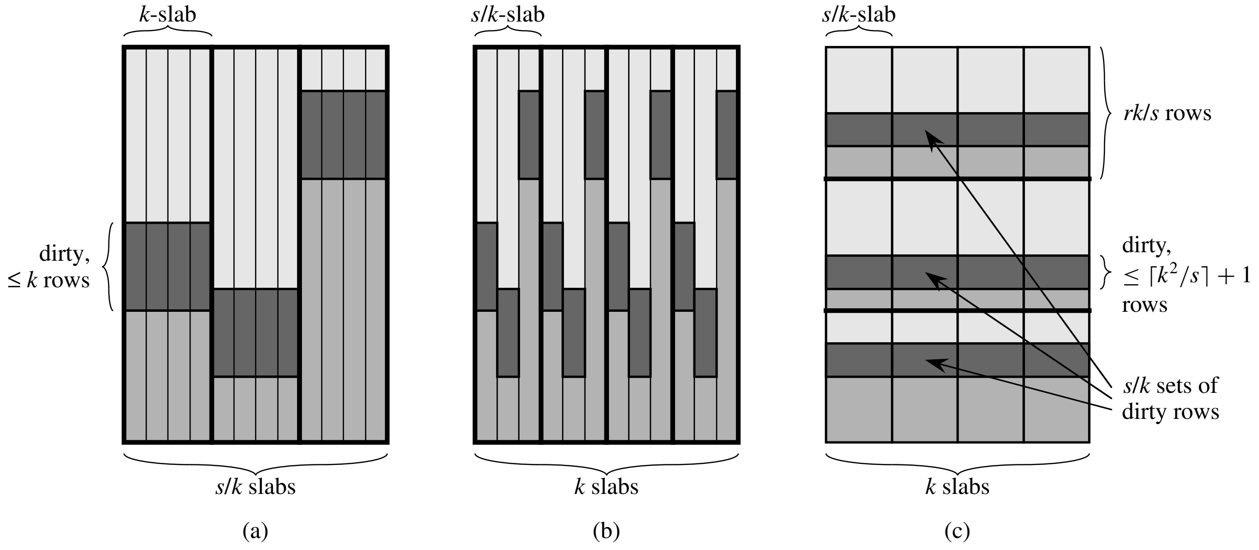


Figure 8: The mesh after steps 3, 4, and 5 of slabpose columnsort. **(a)** After step 3, there are s/k k-slabs, and each k-slab has a dirty area at most k rows high. The dirty areas do not necessarily align among k-slabs. **(b)** After step 4, there are k (s/k)-slabs. If we look at the jth column of each slab, the dirty area is confined to the same set of rows. **(c)** After step 5, within each set of rk/s consecutive rows, the dirty rows are confined to the same set of $\lceil k^2/s \rceil + 1$ rows.

then $\lceil k^2/s \rceil = 1$, and so $(2s^2/k)(\lceil k^2/s \rceil + 1) = 4s^2/k$. This expression is minimized for the largest value of k, which we have required to be at most $\sqrt{s}$. On the other hand, if $k > \sqrt{s}$, let us assume that s is a divisor of k^2. Then $(2s^2/k)(\lceil k^2/s \rceil + 1) = (2s^2/k)(k^2/s + 1) = 2sk + 2s^2/k$. This expression is minimized when $k = \sqrt{s}$.

6 Conclusion

We have seen how to mitigate the divisibility and height restrictions of columnsort. For the original columnsort algorithm, the divisibility restriction is more an artifact of earlier proofs than it is an actual requirement for correctness. We have seen two modifications to columnsort that reduce the height restriction to $r \geq 4s^{3/2}$, subject to the divisibility restriction, and in subblock columnsort we can eliminate the divisibility restriction at the cost of tightening the height restriction to $r \geq 6s^{3/2}$.

What do these results mean in a practical sense? A typical implementation of columnsort on a parallel computer maps columns to the P processors, with N/P values of the mesh mapped to each processor. For an in-core sort, $r = N/P$, and for an out-of-core sort, r is a divisor of N/P. Removing the divisibility restriction means that the number of columns need not be a divisor of either r or N/P. In other words, one restriction on the problem size in terms of the memory per processor is removed.

Relaxing the height restriction increases the maximum problem size that columnsort can handle for a given column size r. Substituting N/r for s in the height restriction $r \geq 2s^2$, the original columnsort algorithm has a problem-size bound of $N \leq r^{3/2}/\sqrt{2}$. When we use the relaxed height restriction of $r \geq 4s^{3/2}$, the problem-size bound increases to $N \leq r^{5/3}/4^{2/3}$, so that r's exponent goes from 3/2 to 5/3.

As mentioned in Section 1, subblock columnsort and slabpose columnsort were devised during the authors' work on out-of-core sorting. The authors have implemented subblock columnsort; details and experimental results on a cluster appear in [CHC03]. An implementation of slabpose columnsort on a cluster is forthcoming.

Acknowledgments

The observation that the original columnsort algorithm does not require the divisibility restriction was first made by Joe Morales. In an undergraduate algorithms course, Joe was given an exam question that asked him to show that without the divisibility restriction, the height restriction of $r \geq 4s^2$ was sufficient to prove correctness. Joe found that $r \geq 2s^2$ always sufficed in any example that he tried. The proof presented here is by the authors.

References

[CC02] Geeta Chaudhry and Thomas H. Cormen. Getting more from out-of-core columnsort. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX 02)*, pages 143–154, January 2002.

[CCW01] Geeta Chaudhry, Thomas H. Cormen, and Leonard F. Wisniewski. Columnsort lives! An efficient out-of-core sorting program. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 2001.

[CHC03] Geeta Chaudhry, Elizabeth A. Hamon, and Thomas H. Cormen. Relaxing the problem-size bound for out-of-core columnsort. Submitted to SPAA 2003, January 2003.

[Lei85] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[MG88] John M. Marberg and Eli Gafni. Sorting in constant number of row and column phases on a mesh. *Algorithmica*, 3:561–572, 1988.

[SS86] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 255–263, May 1986.